Search Strategy and Search Control

- Parallel:
  - Explore all possible trees in parallel
- Depth-first search:
  - Agenda of search states: expand search space incrementally, exploring most recently generated state (tree) each time
  - When you reach a state (tree) inconsistent with input, backtrack to most recent unexplored state (tree)
- Which node to expand?
  - Leftmost
- Which grammar rule to use?
  - Order in the grammar
Basic Algorithm for Top-Down, Depth-First, Left-Right Strategy

- Initialize agenda with ‘S’ tree and point to first word and make this current search state (cur)
- Loop until successful parse or empty agenda
  - Apply all applicable grammar rules to leftmost unexpanded node of cur
    - If this node is a POS category and matches that of the current input, push this onto agenda
    - Else, push new trees onto agenda
  - Pop new cur from agenda

Basic Top-Down Parser

```plaintext
function TOP-DOWN-PARSE(agenda, grammar) returns a parse tree
agenda ← (Initial S tree. Beginning of input)
current-search-state ← POP(agenda)

loop
  if SUCCESSFUL-PARSE?(current-search-state) then
    return TREE(current-search-state)
  else
    if CAT_NODE_TO_EXPAND(current-search-state) is a POS then
      if CAT(node-to-expand) ⊆ POS(current-input current-search-state) then
        PUSH(APPLY-LEXICAL-RULE(current-search-state), agenda)
      else
        return reject
    else
      PUSH(APPLY-RULES(current-search-state, grammar), agenda)
      if agenda is empty then
        return reject
      else
        current-search-state ← NEXT(agenda)
  end
```

Top-Down Depth-First Derivation Using Grammar for $L_1$

Example: Does this flight include a meal?
Book the flight to Houston: Top-Down vs. Bottom-Up

S → VP
VP → Verb Nom
Verb → book
Nom → Houston
Nom → flight
Nom → Art Nom
Art → the
Nom → Art Nom PP
VP → VP PP
PP → Prep Nom
Prep → to

Let’s try both Top-Down and Bottom-Up
Book the flight to Houston: Top-Down Parse (1), (2)

S → VP
VP → Verb Nom
Verb → book
Nom → Houston
Nom → flight
Nom → Art Nom
Art → the
Nom → Art Nom PP
VP → VP PP
PP → Prep Nom
Prep → to

Book the flight to Houston: Top-Down Parse (3), (4)

S → VP
VP → Verb Nom
Verb → book
Nom → Houston
Nom → flight
Nom → Art Nom
Art → the
Nom → Art Nom PP
VP → VP PP
PP → Prep Nom
Prep → to
Book the flight to Houston: Top-Down Parse (5), (6)

S → VP
VP → Verb Nom
Verb → book
Nom → Houston
Nom → flight
Nom → Art Nom
Art → the
Nom → Art Nom PP
VP → VP PP
PP → Prep Nom
Prep → to

Continuing on a few more steps …

Book the flight to Houston: Top-Down Parse (7)

S → VP
VP → Verb Nom
Verb → book
Nom → Houston
Nom → flight
Nom → Art Nom
Art → the
Nom → Art Nom PP
VP → VP PP
PP → Prep Nom
Prep → to
Book the flight to Houston: Final Top-Down Parse Tree

Book the flight to Houston: Bottom-Up (Ply 1 and Ply 2)

IS THIS CORRECT?

Now let’s try bottom-up …
Book the flight to Houston: Bottom-Up (Ply 3 and Ply 4)

Book the flight to Houston: Bottom-Up (Ply 5)

Continuing on one more step …
**Book the flight to Houston: Final Bottom-Up Parse Tree**

**IS THIS CORRECT?**

**Augmenting Top-Down Parsing with Bottom-Up Filtering**

- We saw: Top-Down, depth-first, L-to-R parsing
  - Expands non-terminals along the tree’s left edge down to leftmost leaf of tree
  - Moves on to expand down to next leftmost leaf…
- In a successful parse, current input word will be the first word in derivation of the unexpanded node that the parser is currently processing
- So….lookahead to left-corner of the tree in
  - B is a left-corner of A if A =*=> Bα
  - Build table with left-corners of all non-terminals in grammar and consult before applying rule
Left Corners

Left-Corner Table Using Grammar for $L_1$

<table>
<thead>
<tr>
<th>Category</th>
<th>Left Corners</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Det, PropN, Aux, V</td>
</tr>
<tr>
<td>NP</td>
<td>Det, PropN</td>
</tr>
<tr>
<td>Nom</td>
<td>N</td>
</tr>
<tr>
<td>VP</td>
<td>V</td>
</tr>
</tbody>
</table>

Previous Example:
**Summing Up Parsing Strategies**

- Parsing is a search problem which may be implemented with many search strategies
- Top-Down vs. Bottom-Up Parsers
  - Both generate too many useless trees
  - Combine the two to avoid over-generation: Top-Down Parsing with Bottom-Up look-ahead
- Left-corner table provides more efficient look-ahead
  - Pre-compute all POS that can serve as the leftmost POS in the derivations of each non-terminal category

**Three Critical Problems in Parsing**

- Left Recursion
- Ambiguity
- Repeated Parsing of Sub-trees
Left Recursion

Depth-first search will never terminate if grammar is left recursive: $A \rightarrow A \ B \ \beta$

**Examples:**
- $NP \rightarrow NP \ PP$, $VP \rightarrow VP \ PP$, $S \rightarrow S$ and $S$, $NP \rightarrow NP$ and $NP$

**Indirect Left Recursion:** $A^* \rightarrow A \ B \ \beta$

**Example:** $NP \rightarrow$ Det Nominal, $Det \rightarrow NP$ ‘s

Solutions to Left Recursion

- Rule ordering
- Don't use recursive rules
- Limit depth of recursion in parsing to some analytically or empirically set limit
- Don't use top-down parsing
Rule Ordering

Bad:
- NP → NP PP
- NP → Det Nominal

Better alternative:
- First: NP → Det Nominal
- Then: NP → NP PP

Grammar Rewriting

- Rewrite left-recursive grammar as weakly equivalent non-recursive one.
- Can be done:
  - By Hand (ick) or …
  - Automatically
- Example
  Rewrite: NP → NP PP, NP → Det Nominal
  As: NP → Det Nominal Stuff, Stuff → PP Stuff, Stuff → ε
Problems with Grammar Rewriting

• Original:
  [NP [NP the book]
    [PP on [NP [NP the table]
      [PP in [NP [NP the yard]
        [PP of [NP the house]]]]]]

• Becomes:
  [NP the book
   [Stuff
     [PP on [NP the table
       [Stuff
         [PP in [NP the yard
           [Stuff
             [PP of [NP the house [Stuff]]]
           [Stuff]]]
         [Stuff]]]]
   [Stuff]]]
Ambiguity

Local Ambiguity
– Leads to hypotheses that are locally reasonable but eventually lead nowhere
– “Book that flight”

Global Ambiguity
– Leads to multiple parses for the same input
– “I shot an elephant in my pajamas”

More Ambiguity Examples

Multiple legal structures
– Attachment (e.g. I saw a man on a hill with telescope)
– Coordination (e.g. younger cats and dogs)
– NP bracketing (e.g. Spanish language teachers)
Two Parse Trees for Ambiguous Sentence

More Ambiguity: ‘Can you book TWA flights?’
A Correct Parse for ‘Show me the meal on Flight UA 386 from San Francisco to Denver’

Inefficient Re-Parsing of Subtrees
**Invariants**

- Despite ambiguity, there are invariants
- Sub-components of final parse tree are re-analyzed unnecessarily
- Except for top-most component, every part of final tree is derived more than once.

**What’s the solution?**

- Key to efficient processing is reuse
- Fill table with solutions to sub-problems for later use.
- We want an algorithm that:
  - Does not do repeated work
  - Does top-down search with bottom-up filtering
  - Solves left-recursion problem
  - Solves an exponential problem
Dynamic Programming and Parsing

- Create table of solutions to sub-problems (e.g. subtrees) as parse proceeds
- Look up subtrees for each constituent rather than re-parsing
- Since all parses implicitly stored, all available for later disambiguation

CYK Algorithm

- Classic example of dynamic programming paradigm
- Bottom-up CFG parsing
- Requires grammar in Chomsky Normal Form (CNF): binary branching.
  - $A \rightarrow BC$
  - $A \rightarrow w$
- Q: Is this strongly or weakly equivalent to original grammar?
- Fill in upper-triangular matrix (chart) in bottom-up fashion using CNF grammar, where $\text{chart}(i,j) = \{A \mid A \rightarrow w_{i+1} \ldots w_j\}$
- Note that cells include sets of non-terminals.
Conversion to CNF

* Procedure:
  - Copy all conforming rules unchanged
  - Convert terminals w/in rules to dummy non-terminals
  - Convert unit productions (this is a flattening operation!)
  - Binarize rules
* Example:
  - $S \rightarrow VP$
  - $VP \rightarrow \text{Verb}_{trans} NP$
  - $VP \rightarrow V NP PP$
  - $\text{INF}-VP \rightarrow to VP$
* Rule 4 becomes 4' and 4'':
  - $\text{INF}-VP \rightarrow TO VP$
  - $TO \rightarrow to$
* Rule 1 becomes 1' and 1'':
  - $S \rightarrow \text{Verb}_{trans} NP$
  - $S \rightarrow V NP PP$
* Rules 3 and 1'' become:
  - $S \rightarrow V \text{Double-Comp}$
  - $VP \rightarrow V \text{Double-Comp}$
  - $\text{Double-Comp} \rightarrow NP PP$

Example of Conversion to CNF

<table>
<thead>
<tr>
<th>OLD</th>
<th>NEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow NP VP'$</td>
<td>$S \rightarrow NP VP$</td>
</tr>
<tr>
<td>$S \rightarrow \text{Name P P P}$</td>
<td>$S \rightarrow \text{Name P P P}$</td>
</tr>
<tr>
<td>$S \rightarrow VP$</td>
<td>$S \rightarrow \text{book</td>
</tr>
<tr>
<td>$NP \rightarrow \text{Proper Noun}$</td>
<td>$NP \rightarrow \text{Proper Noun}$</td>
</tr>
<tr>
<td>$NP \rightarrow \text{Det Nominal}$</td>
<td>$NP \rightarrow \text{Det Nominal}$</td>
</tr>
<tr>
<td>$\text{Nominal} \rightarrow \text{Nominal Noun}$</td>
<td>$\text{Nominal} \rightarrow \text{Nominal Noun}$</td>
</tr>
<tr>
<td>$\text{Nominal} \rightarrow \text{Nominal P P}$</td>
<td>$\text{Nominal} \rightarrow \text{Nominal P P}$</td>
</tr>
<tr>
<td>$VP \rightarrow \text{Verb}$</td>
<td>$VP \rightarrow \text{Verb}$</td>
</tr>
<tr>
<td>$VP \rightarrow \text{Verb NP}$</td>
<td>$VP \rightarrow \text{Verb NP}$</td>
</tr>
<tr>
<td>$VP \rightarrow \text{Verb NP P P}$</td>
<td>$VP \rightarrow \text{Verb NP P P}$</td>
</tr>
<tr>
<td>$PP \rightarrow \text{Preposition P P}$</td>
<td>$PP \rightarrow \text{Preposition P P}$</td>
</tr>
</tbody>
</table>

Figure 13.8: $A_1$ Grammar and its conversion to CNF. Note that although they aren’t shown here all the original lexical entries from $A_1$ carry over unchanged as well.
**CYK Algorithm Code**

```plaintext
function CKY-Parse(words, grammar) returns table

for j ← from 1 to LENGTH(words) do
  table[j-1, j] ← {A | A → words[j] ∈ grammar }

for i ← from j – 2 downto 0 do
  for k ← i + 1 to j – 1 do
    table[i,j] ← table[i,j] ∪ {A | A → B C ∈ grammar,
                                 B ∈ table[i,k],
                                 C ∈ table[k,j] }
```

**Completed Parse Table for Book the flight through Houston**

```
      Book   the   flight   through   Houston
    0.1 S,VP,Verb  Nominal, Noun
    0.2   Det
    0.3   NP  
    0.4   Nominal, Noun
    0.5   Prep
    0.6   NP, Proper Noun

  [1.2] [1.3] [1.4] [1.5] [2.3] [2.4] [2.5] [3.4] [3.5] [4.5]
```
<table>
<thead>
<tr>
<th>Book</th>
<th>the</th>
<th>flight</th>
<th>through</th>
<th>Houston</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S.V.P., Verb</td>
<td>Det</td>
<td>Nominal, Noun</td>
<td>[0, 2]</td>
</tr>
<tr>
<td>2</td>
<td>S.V.P., X2</td>
<td>Nominal, Noun</td>
<td>Prep</td>
<td>[1, 3]</td>
</tr>
<tr>
<td></td>
<td>NP</td>
<td>[2, 3]</td>
<td>[2, 4]</td>
<td>[2, 5]</td>
</tr>
</tbody>
</table>

Filling last column after reading the word Houston
Filling last column after reading the word Houston

<table>
<thead>
<tr>
<th>Book</th>
<th>the</th>
<th>flight</th>
<th>through</th>
<th>Houston</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Filling last column after reading the word Houston

This indicates the presence of three alternative parses for this input:
- PP modifies flight
- PP modifies book
- PP is an argument of the verb book

Is this a parser or a recognizer?
Can use these “backpointers” to turn recognizer into a “parser”

How does Dynamic Programming Provide Savings?

Multiple ways to fill cells with the same non-terminal—but only one copy is included in the cell.
Disadvantages of CYK

- Primary disadvantage: Conversion to CNF.
  - Leads to “weak equivalence” w.r.t. original grammar
  - Introduces non-linguistically relevant categories e.g., X2
  - Complicates semantic analysis
  - Adds a level of “hidden” complexity. Parse time is $Gn^3$ (where $G$ is size of grammar) but conversion to CNF throws in another factor of $G$: $G^2n^3$ — same as Earley!

- Is there a parsing approach that adopts dynamic programming and accepts arbitrary CFGs?

Earley Algorithm

- Uses dynamic programming to do parallel top-down search in (worst case) $O(N^3)$ time
- First, left-to-right pass fills out a chart with $N+1$ states
  - Think of chart entries as sitting between words in the input string keeping track of states of the parse at these positions
  - For each word position, chart contains set of states representing all partial parse trees generated to date. E.g. chart[0] contains all partial parse trees generated at the beginning of the sentence
Chart Entries

Represent three types of constituents:

- predicted constituents
- in-progress constituents
- completed constituents

Progress in parse represented by Dotted Rules

- Position of • indicates type of constituent
- 0 Book 1 that 2 flight 3
  - S → • VP, [0,0] (predicted)
  - NP → Det • Nominal, [1,2] (in progress)
  - VP → V NP •, [0,3] (completed)
- [x,y] tells us what portion of the input is spanned so far by this rule
- Each State $s_i$:
  <dotted rule>, [<back pointer>,<current position>]
**Book 1 that flight 3**

- **S -> • VP, [0,0]**
  - First 0 means S constituent begins at the start of input
  - Second 0 means the dot here too
  - So, this is a top-down prediction
- **NP -> Det • Nominal, [1,2]**
  - the NP begins at position 1
  - the dot is at position 2
  - so, Det has been successfully parsed
  - Nom predicted next
- **VP -> V NP •, [0,3]**
  - Successful VP parse of entire input

---

**Successful Parse**

- Final answer found by looking at last state set in chart
- If entry contains S -> α • [0,N] then input parsed successfully
- Chart will also contain record of all possible parses of input string, given the grammar

| Chart[1] | S12 Verb -> book • [0,1] Scanner |
| Chart[2] | S23 Det -> that • [1,2] Scanner |
|        | S29 Nominal -> Noun • [2,3] (S28) 
|        | S30 NP -> Det Nominal • [1,3] (S23, S29) 
|        | S33 VP -> Verb NP • [0,3] (S12, S30) 
|        | S36 S -> VP • [0,3] (S33) |
Parsing Procedure for the Earley Algorithm

- Move through each set of states in order, applying one of three operators to each state:
  - predictor: add predictions to the chart
  - scanner: read input and add corresponding state to chart
  - completer: move dot to right when new constituent found
- Results (new states) added to current or next set of states in chart
- No backtracking and no states removed: keep complete history of parse

States and State Sets

- **Dotted Rule** $s_i$ represented as $<\text{dotted rule}>, [<\text{back pointer}>, <\text{current position}>]$:
- **State Set** $S_j$ to be a collection of states $s_i$ with the same $<\text{current position}>$. 
Earley Algorithm (simpler!)

1. Add Start → • S, [0,0] to state set 0

2. **Predict** all states you can, adding new predictions to state set 0. Let i = 1.

3. **Scan** input word i—add all matched states to state set $S_i$. Add all new states produced by **Complete** to state set $S_i$. Add all new states produced by **Predict** to state set $S_i$. Unless $i=n$, (a) Let $i = i + 1$; (b) repeat step 3.

4. At the end, see if state set $n$ contains Start → S • , [0,n]
3 Main Sub-Routines of Earley Algorithm

- Predictor: Adds predictions into the chart.
- Completer: Moves the dot to the right when new constituents are found.
- Scanner: Reads the input words and enters states representing those words into the chart.

Predictor

- Intuition: create new state for top-down prediction of new phrase.
- Applied when non part-of-speech non-terminals are to the right of a dot: \( S \rightarrow \bullet \text{VP [0,0]} \)
- Adds new states to current chart
  - One new state for each expansion of the non-terminal in the grammar
    \( \text{VP} \rightarrow \bullet \text{V [0,0]} \)
    \( \text{VP} \rightarrow \bullet \text{V NP [0,0]} \)
- Formally:
  \( S_i: A \rightarrow \alpha \cdot B \beta, [i,j] \)
  \( S_j: B \rightarrow \cdot \gamma, [j,j] \)
**Scanner**

- Intuition: Create new states for rules matching part of speech of next word.
- Applicable when a word is to the right of a dot: \( V \rightarrow \cdot \text{book} \ [0,0] \) ‘Book…’
- Looks at current word in input
- If match, adds state(s) to **next** chart
  \( V \rightarrow \text{book} \cdot \ [0,1] \)
- Formally:
  \[ S_j: A \rightarrow \alpha \cdot B \beta, [i,j] \]
  \[ S_{j+1}: A \rightarrow \alpha B \cdot \beta, [i,j+1] \]

**Completer**

- Intuition: parser has finished a new phrase, so must find and advance states all that were waiting for this
- Applied when dot has reached right end of rule
  \( \text{NP} \rightarrow \text{Det Nom} \cdot [1,3] \)
- Find all states w/dot at 1 and expecting an NP
  \( \text{NP} \rightarrow \text{V \cdot NP} \ [0,1] \)
- Adds new (completed) state(s) to **current** chart
  \( \text{VP} \rightarrow \text{V \ NP} \cdot [0,3] \)
- Formally:
  \[ S_k: B \rightarrow \delta \cdot, [j,k] \]
  \[ S_k: A \rightarrow \alpha B \cdot \beta, [i,k], \]
  \[ \text{where: } S_j: A \rightarrow \alpha \cdot B \beta, [i,j]. \]
Example: State Set $S_0$ for Parsing “Book that flight” using Grammar for $L_1$

<table>
<thead>
<tr>
<th>Chart[0]</th>
<th>S0</th>
<th>$\gamma \rightarrow \bullet S$</th>
<th>[0.0] Dummy start state</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>$S \rightarrow \bullet NP VP$</td>
<td>[0.0] Predictor</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>$S \rightarrow \bullet Aux NP VP$</td>
<td>[0.0] Predictor</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>$S \rightarrow \bullet VP$</td>
<td>[0.0] Predictor</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>$NP \rightarrow \bullet Pronoun$</td>
<td>[0.0] Predictor</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>$NP \rightarrow \bullet Proper-Noun$</td>
<td>[0.0] Predictor</td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>$NP \rightarrow \bullet Det Nominal$</td>
<td>[0.0] Predictor</td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td>$VP \rightarrow \bullet Verb$</td>
<td>[0.0] Predictor</td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td>$VP \rightarrow \bullet Verb NP$</td>
<td>[0.0] Predictor</td>
<td></td>
</tr>
<tr>
<td>S9</td>
<td>$VP \rightarrow \bullet Verb NP PP$</td>
<td>[0.0] Predictor</td>
<td></td>
</tr>
<tr>
<td>S10</td>
<td>$VP \rightarrow \bullet Verb PP$</td>
<td>[0.0] Predictor</td>
<td></td>
</tr>
<tr>
<td>S11</td>
<td>$VP \rightarrow \bullet VP PP$</td>
<td>[0.0] Predictor</td>
<td></td>
</tr>
</tbody>
</table>

Example: State Set $S_1$ for Parsing “Book that flight”

<table>
<thead>
<tr>
<th>Chart[1]</th>
<th>S12</th>
<th>Verb $\rightarrow$ book $\bullet$</th>
<th>[0.1] Scanner</th>
</tr>
</thead>
<tbody>
<tr>
<td>S13</td>
<td>$VP \rightarrow$ Verb $\bullet$</td>
<td>[0.1] Completer</td>
<td></td>
</tr>
<tr>
<td>S14</td>
<td>$VP \rightarrow$ Verb $\bullet NP$</td>
<td>[0.1] Completer</td>
<td></td>
</tr>
<tr>
<td>S15</td>
<td>$VP \rightarrow$ Verb $\bullet NP PP$</td>
<td>[0.1] Completer</td>
<td></td>
</tr>
<tr>
<td>S16</td>
<td>$VP \rightarrow$ Verb $\bullet PP$</td>
<td>[0.1] Completer</td>
<td></td>
</tr>
<tr>
<td>S17</td>
<td>$S \rightarrow$ VP $\bullet$</td>
<td>[0.1] Completer</td>
<td></td>
</tr>
<tr>
<td>S18</td>
<td>$VP \rightarrow$ VP $\bullet PP$</td>
<td>[0.1] Completer</td>
<td></td>
</tr>
<tr>
<td>S19</td>
<td>$NP \rightarrow$ Pronoun</td>
<td>[1.1] Predictor</td>
<td></td>
</tr>
<tr>
<td>S20</td>
<td>$NP \rightarrow$ Proper-Noun</td>
<td>[1.1] Predictor</td>
<td></td>
</tr>
<tr>
<td>S21</td>
<td>$NP \rightarrow$ Det Nominal</td>
<td>[1.1] Predictor</td>
<td></td>
</tr>
<tr>
<td>S22</td>
<td>$PP \rightarrow$ Prep NP</td>
<td>[1.1] Predictor</td>
<td></td>
</tr>
</tbody>
</table>
Prediction of Next Rule

- When VP $\rightarrow$ Verb • is itself processed by the Completer, S $\rightarrow$ VP • is added to Chart[1] since VP is a left corner of S
- Last 4 rules in Chart[1] are added by **Predictor** when VP $\rightarrow$ Verb • NP is processed
- And so on….

Last Two States

<table>
<thead>
<tr>
<th>Chart[2]</th>
<th>Rule</th>
<th>Predicted</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S23</td>
<td>Det $\rightarrow$ that •</td>
<td>[1,2]</td>
<td>Scanner</td>
</tr>
<tr>
<td>S24</td>
<td>NP $\rightarrow$ Det • Noun</td>
<td>[1,2]</td>
<td>Completer</td>
</tr>
<tr>
<td>S25</td>
<td>Noun $\rightarrow$ • Noun</td>
<td>[2,2]</td>
<td>Predictor</td>
</tr>
<tr>
<td>S26</td>
<td>Noun • Noun Noun</td>
<td>[2,2]</td>
<td>Predictor</td>
</tr>
<tr>
<td>S27</td>
<td>Noun • Noun PP</td>
<td>[2,2]</td>
<td>Predictor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chart[3]</th>
<th>Rule</th>
<th>Predicted</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S28</td>
<td>Noun $\rightarrow$ flight •</td>
<td>[2,3]</td>
<td>Scanner</td>
</tr>
<tr>
<td>S29</td>
<td>Noun • Noun</td>
<td>[2,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S30</td>
<td>NP $\rightarrow$ Det Noun •</td>
<td>[2,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S31</td>
<td>Noun • Noun Noun</td>
<td>[2,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S32</td>
<td>Noun • Noun PP</td>
<td>[2,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S33</td>
<td>VP $\rightarrow$ Verb NP •</td>
<td>[0,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S34</td>
<td>VP $\rightarrow$ Verb NP • PP</td>
<td>[0,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S35</td>
<td>PP $\rightarrow$ • Prep NP</td>
<td>[3,3]</td>
<td>Predictor</td>
</tr>
<tr>
<td>S36</td>
<td>S $\rightarrow$ VP •</td>
<td>[0,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S37</td>
<td>VP $\rightarrow$ VP • PP</td>
<td>[0,3]</td>
<td>Completer</td>
</tr>
</tbody>
</table>

Are we done? **YES!**
How do we retrieve the parses at the end?

- Augment the Completer to add pointers to prior states it advances as a field in the current state
  - i.e. what state did we advance here?
  - Read the pointers back from the final state

<table>
<thead>
<tr>
<th>Chart[1]</th>
<th>S12 Verb → book •</th>
<th>[0,1]</th>
<th>Scanner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart[2]</td>
<td>S23 Det → that •</td>
<td>[1,2]</td>
<td>Scanner</td>
</tr>
<tr>
<td></td>
<td>S29 Nominal → Noun •</td>
<td>[2,3]</td>
<td>(S28)</td>
</tr>
<tr>
<td></td>
<td>S30 NP → Det Nominal •</td>
<td>[1,3]</td>
<td>(S23, S29)</td>
</tr>
<tr>
<td></td>
<td>S33 VP → Verb NP •</td>
<td>[0,3]</td>
<td>(S12, S30)</td>
</tr>
<tr>
<td></td>
<td>S36 S → VP •</td>
<td>[0,3]</td>
<td>(S33)</td>
</tr>
</tbody>
</table>

Error Handling

- What happens when we look at the contents of the last table column and don't find a $S \rightarrow \alpha \bullet$ rule?
  - Is it a total loss? No...
  - Chart contains every constituent and combination of constituents possible for the input given the grammar

- Also useful for partial parsing or shallow parsing used in information extraction
Earley’s Keys to Efficiency

- Left-recursion, Ambiguity and repeated re-parsing of subtrees
  - Solution: dynamic programming
- Combine top-down predictions with bottom-up look-ahead to avoid unnecessary expansions
- Earley is still one of the most efficient parsers
- All efficient parsers avoid re-computation in a similar way.
- But Earley doesn’t require grammar conversion!

Next Time

- Work on midterm
- Read Chapter 4